



下载APP



26 | 方法：如何用类型嵌入模拟实现“继承”？

2021-12-13 Tony Bai

《Tony Bai · Go语言第一课》

[课程介绍 >](#)**讲述：Tony Bai**

时长 23:13 大小 21.28M



你好，我是 Tony Bai。

在前面的两节课中，我们学习了 Go 方法的声明、本质，以及 receiver 类型选择的三个原则。可以说，学完这些内容，我们就基本上解决了**独立的自定义类型**的方法的设计问题。

什么是**独立的自定义类型**呢？就是这个类型的所有方法都是自己显式实现的。我们举个例子，自定义类型 T 有两个方法 M1 和 M2，如果 T 是一个独立的自定义类型，那我们在声明类型 T 的 Go 包源码文件中一定可以找到其所有方法的实现代码，比如：



```
1 func (T) M1() {...}
2 func (T) M2() {...}
```

[复制代码](#)

这里你一定会问：**难道还有某种自定义类型的方法不是自己显式实现的吗？**当然有！这就是我们这讲中要重点讲解的内容：**如何让某个自定义类型“继承”其他类型的方法实现。**

这里你肯定又会提出质疑：**老师，你不是说过 Go 不支持经典的面向对象编程范式吗？怎么还会有继承这一说法呢？**没错！Go 语言从设计伊始，就决定不支持经典面向对象的编程范式与语法元素，所以我们这里只是借用了“继承”这个词汇而已，说是“继承”，实则依旧是一种**组合**的思想。

而这种“继承”，我们是通过 Go 语言的**类型嵌入（Type Embedding）**来实现的。所以这一节课，我们就来学习一下这种语法，看看通过这种语法，我们如何实现对嵌入类型的方法的“继承”，同时也搞清楚这种方式对新定义的类型的方法集合的影响。

现在，我们先来学习一下什么是类型嵌入。

什么是类型嵌入

类型嵌入指的就是在一个类型的定义中嵌入了其他类型。Go 语言支持两种类型嵌入，分别是**接口类型的类型嵌入**和**结构体类型的类型嵌入**。

接口类型的类型嵌入

我们先用一个案例，直观地了解一下什么是接口类型的类型嵌入。虽然我们还没有系统学习接口类型，但在前面的讲解中，我们已经多次接触了接口类型。我们知道，**接口类型声明了由一个方法集合代表的接口**，比如下面接口类型 E：

```
1 type E interface {  
2     M1()  
3     M2()  
4 }
```

 复制代码

这个接口类型 E 的方法集合，包含两个方法，分别是 M1 和 M2，它们组成了 E 这个接口类型所代表的接口。如果某个类型实现了方法 M1 和 M2，我们就说这个类型实现了 E 所代表的接口。

此时，我们再定义另外一个接口类型 `I`，它的方法集合中包含了三个方法 `M1`、`M2` 和 `M3`，如下面代码：

[复制代码](#)

```
1 type I interface {  
2     M1()  
3     M2()  
4     M3()  
5 }
```

我们看到接口类型 `I` 方法集合中的 `M1` 和 `M2`，与接口类型 `E` 的方法集合中的方法完全相同。在这种情况下，我们可以**用接口类型 `E` 替代上面接口类型 `I` 定义中 `M1` 和 `M2`**，如下面代码：

[复制代码](#)

```
1 type I interface {  
2     E  
3     M3()  
4 }
```

像这种在一个接口类型（`I`）定义中，嵌入另外一个接口类型（`E`）的方式，就是我们说的**接口类型的类型嵌入**。

而且，这个带有类型嵌入的接口类型 `I` 的定义与上面那个包含 `M1`、`M2` 和 `M3` 的接口类型 `I` 的定义，是等价的。因此，我们可以得到一个结论，这种**接口类型嵌入的语义就是新接口类型（如接口类型 `I`）将嵌入的接口类型（如接口类型 `E`）的方法集合，并入到自己的方法集合中**。

到这里你可能会问，我在接口类型定义中平铺方法列表就好了，为啥要使用类型嵌入方式定义接口类型呢？其实**这也是 Go 组合设计哲学的一种体现**。

按 Go 语言惯例，Go 中的接口类型中只包含少量方法，并且常常只是一个方法。通过在接口类型中嵌入其他接口类型可以实现接口的组合，这也是 **Go 语言中基于已有接口类型构建新接口类型的惯用法**。

我们在 Go 标准库中可以看到很多这种组合方式的应用，最常见的莫过于 io 包中一系列接口的定义了。比如，io 包的 `Reader`、`ReadWriteCloser` 等接口类型就是通过嵌入 `Reader`、`Writer` 或 `Closer` 三个基本的接口类型组合而成的。下面是仅包含单一方法的 io 包 `Reader`、`Writer` 和 `Closer` 的定义：

 复制代码

```
1 // $GOROOT/src/io/io.go
2
3 type Reader interface {
4     Read(p []byte) (n int, err error)
5 }
6
7 type Writer interface {
8     Write(p []byte) (n int, err error)
9 }
10
11 type Closer interface {
12     Close() error
13 }
```

下面的 io 包的 `Reader`、`ReadWriteCloser` 等接口类型，通过嵌入上面基本接口类型组合而形成：

 复制代码

```
1 type ReadWriter interface {
2     Reader
3     Writer
4 }
5
6 type ReadCloser interface {
7     Reader
8     Closer
9 }
10
11 type WriteCloser interface {
12     Writer
13     Closer
14 }
15
16 type ReadWriteCloser interface {
17     Reader
18     Writer
19     Closer
20 }
```

不过，这种通过嵌入其他接口类型来创建新接口类型的方式，在 Go 1.14 版本之前是有**约束**的：如果新接口类型嵌入了多个接口类型，这些嵌入的接口类型的方法集合不能有交集，同时嵌入的接口类型的方法集合中的方法名字，也不能与新接口中的其他方法同名。比如我们用 **Go 1.12.7 版本** 运行下面例子，Go 编译器就会报错：

 复制代码

```
1 type Interface1 interface {
2     M1()
3 }
4
5 type Interface2 interface {
6     M1()
7     M2()
8 }
9
10 type Interface3 interface {
11     Interface1
12     Interface2 // Error: duplicate method M1
13 }
14
15 type Interface4 interface {
16     Interface2
17     M2() // Error: duplicate method M2
18 }
19
20 func main() {
21 }
```

我们具体看一下例子中的两个编译报错：第一个是因为 Interface3 中嵌入的两个接口类型 Interface1 和 Interface2 的方法集合有交集，交集是方法 M1；第二个报错是因为 Interface4 类型中的方法 M2 与嵌入的接口类型 Interface2 的方法 M2 重名。

但自 Go 1.14 版本开始，Go 语言去除了这些约束，我们使用 Go 1.17 版本运行上面这个示例就不会得到编译错误了。

当然，接口类型的类型嵌入比较简单，我们只要把握好它的语义，也就是“方法集合并入”就可以了。结构体类型的类型嵌入就要更复杂一些了，接下来我们一起来学习一下。

结构体类型的类型嵌入

我们在第 17 讲中对 Go 结构体类型进行了系统的讲解，在那一讲中我们遇到的结构体都是类似下面这样的：

 复制代码

```
1 type S struct {
2     A int
3     b string
4     c T
5     p *P
6     _ [10]int8
7     F func()
8 }
```

结构体类型 S 中的每个字段（field）都有唯一的名字与对应的类型，即便是使用空标识符占位的字段，它的类型也是明确的，但这还不是 Go 结构体类型的“完全体”。Go 结构体类型定义还有另外一种形式，那就是**带有嵌入字段（Embedded Field）的结构体定义**。我们看下面这个例子：

 复制代码

```
1 type T1 int
2 type t2 struct{
3     n int
4     m int
5 }
6
7 type I interface {
8     M1()
9 }
10
11 type S1 struct {
12     T1
13     *t2
14     I
15     a int
16     b string
17 }
```

我们看到，结构体 S1 定义中有三个“非常规形式”的标识符，分别是 T1、t2 和 I，这三个标识符究竟代表的是什么呢？是字段名还是字段的类型呢？这里我直接告诉你答案：它们**既代表字段的名字，也代表字段的类型**。我们分别以这三个标识符为例，说明一下它们的具体含义：

标识符 T1 表示字段名为 T1，它的类型为自定义类型 T1；

标识符 t2 表示字段名为 t2，它的类型为自定义结构体类型 t2 的指针类型；

标识符 I 表示字段名为 I，它的类型为接口类型 I。

这种以某个类型名、类型的指针类型名或接口类型名，直接作为结构体字段的方式就叫做**结构体的类型嵌入**，这些字段也被叫做**嵌入字段 (Embedded Field)**。

那么，嵌入字段怎么用呢？它跟普通结构体字段有啥不同呢？我们结合具体的例子，简单说一下嵌入字段的用法：

 复制代码

```
1  type MyInt int
2
3  func (n *MyInt) Add(m int) {
4      *n = *n + MyInt(m)
5  }
6
7  type t struct {
8      a int
9      b int
10 }
11
12 type S struct {
13     *MyInt
14     t
15     io.Reader
16     s string
17     n int
18 }
19
20 func main() {
21     m := MyInt(17)
22     r := strings.NewReader("hello, go")
23     s := S{
24         MyInt: &m,
25         t: t{
26             a: 1,
27             b: 2,
28         },
29         Reader: r,
30         s:     "demo",
31     }
32
33     var sl = make([]byte, len("hello, go"))
34     s.Reader.Read(sl)
```

```
35     fmt.Println(string(sl)) // hello, go
36     s.MyInt.Add(5)
37     fmt.Println(*s.MyInt) // 22
38 }
```

在分析这段代码之前，我们要先明确一点，那就是嵌入字段的可见性与嵌入字段的类型的可见性是一致的。如果嵌入类型的名字是首字母大写的，那么也就说明这个嵌入字段是可导出的。

现在我们来看这个例子。

首先，这个例子中的结构体类型 `S` 使用了类型嵌入方式进行定义，它有三个嵌入字段 `MyInt`、`t` 和 `Reader`。这里，你可能会问，为什么第三个嵌入字段的名字为 `Reader` 而不是 `io.Reader`？这是因为，Go 语言规定如果结构体使用从其他包导入的类型作为嵌入字段，比如 `pkg.T`，那么这个嵌入字段的字段名就是 `T`，代表的类型为 `pkg.T`。

接下来，我们再来看结构体类型 `S` 的变量的初始化。我们使用 `field:value` 方式对 `S` 类型的变量 `s` 的各个字段进行初始化。和普通的字段一样，初始化嵌入字段时，我们可以直接用嵌入字段名作为 `field`。

而且，通过变量 `s` 使用这些嵌入字段时，我们也可以像普通字段那样直接用变量 `s`+字段选择符 `.`+嵌入字段的名字，比如 `s.Reader`。我们还可以通过这种方式调用嵌入字段的方法，比如 `s.Reader.Read` 和 `s.MyInt.Add`。

这样看起来，嵌入字段的用法和普通字段没啥不同呀？也不完全是，Go 还是对嵌入字段有一些约束的。比如，和 Go 方法的 receiver 的基类型一样，嵌入字段类型的底层类型不能为指针类型。而且，嵌入字段的名字在结构体定义也必须是唯一的，这也意味这如果两个类型的名字相同，它们无法同时作为嵌入字段放到同一个结构体定义中。不过，这些约束你了解一下就可以了，一旦违反，Go 编译器会提示你的。

到这里，我们看到嵌入字段在使用上确实和普通字段没有多大差别，那我们为什么要用嵌入字段这种方式来定义结构体类型呢？别急，我们继续向下看。

“实现继承”的原理

我们将上面例子代码做一下细微改动，我这里只列了变化部分的代码：

 复制代码

```
1 var sl = make([]byte, len("hello, go"))
2 s.Read(sl)
3 fmt.Println(string(sl))
4 s.Add(5)
5 fmt.Println(*(s.MyInt))
```

看到这段代码，你肯定会问：老师，类型 `S` 也没有定义 `Read` 方法和 `Add` 方法啊，这样写会导致 Go 编译器报错的。如果你有这个疑问，可以暂停一下，先用你手头上的 Go 编译器编译运行一下这段代码看看。

惊不惊喜，意不意外？这段程序不但没有引发编译器报错，还可以正常运行并输出与前面例子相同的结果！

这段代码似乎在告诉我们：**Read 方法与 Add 方法就是类型 `S` 方法集合中的方法**。但是，这里类型 `S` 明明没有显式实现这两个方法呀，它是从哪里得到这两个方法的实现的呢？

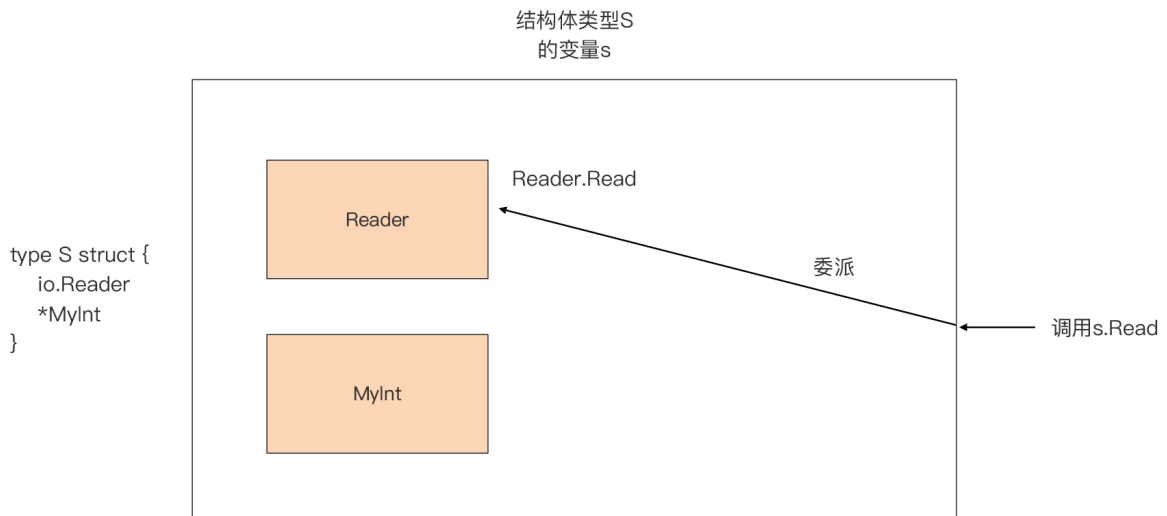
其实，这两个方法就来自结构体类型 `S` 的两个嵌入字段 `Reader` 和 `MyInt`。结构体类型 `S` “继承”了 `Reader` 字段的方法 `Read` 的实现，也“继承”了 `*MyInt` 的 `Add` 方法的实现。注意，我这里的“继承”用了引号，说明这并不是真正的继承，它只是 Go 语言的一种“障眼法”。

这种“障眼法”的工作机制是这样的，当我们通过结构体类型 `S` 的变量 `s` 调用 `Read` 方法时，Go 发现结构体类型 `S` 自身并没有定义 `Read` 方法，于是 Go 会查看 `S` 的嵌入字段对应的类型是否定义了 `Read` 方法。这个时候，`Reader` 字段就被找了出来，之后 `s.Read` 的调用就被转换为 `s.Reader.Read` 调用。

这样一来，嵌入字段 `Reader` 的 `Read` 方法就被提升为 `S` 的方法，放入了类型 `S` 的方法集合。同理 `*MyInt` 的 `Add` 方法也被提升为 `S` 的方法而放入 `S` 的方法集合。从外部来看，这种嵌入字段的方法的提升就给了我们一种结构体类型 `S` “继承”了 `io.Reader` 类型 `Read` 方法的实现，以及 `*MyInt` 类型 `Add` 方法的实现的错觉。

到这里，我们就清楚了，嵌入字段的使用的确可以帮我们在 Go 中实现方法的“继承”。

这节课开始我们就提过，类型嵌入这种看似“继承”的机制，实际上是一种组合的思想。更具体点，它是一种组合中的代理（delegate）模式，如下图所示：



我们看到，S 只是一个代理（delegate），对外它提供了它可以代理的所有方法，如例子中的 Read 和 Add 方法。当外界发起对 S 的 Read 方法的调用后，S 将该调用委派给它内部的 Reader 实例来实际执行 Read 方法。

当然，嵌入字段的类型不同，自定义结构体类型可以代理的方法就不同，那自定义结构体类型究竟可以代理哪些方法呢？换个角度说，嵌入字段对结构体的方法集合有哪些影响呢？下面我们就分情况来看嵌入不同类型的结构体类型的方法集合中，都包含哪些方法。

类型嵌入与方法集合

在前面讲解接口类型的类型嵌入时，我们提到过接口类型的类型嵌入的本质，就是嵌入类型的方法集合并入到新接口类型的方法集合中，并且，接口类型只能嵌入接口类型。而结构体类型对嵌入类型的要求就比较宽泛了，可以是任意自定义类型或接口类型。

下面我们就分别看看，在这两种情况下，结构体类型的方法集合会有怎样的变化。我们依旧借助上一讲中的 dumpMethodSet 函数来输出各个类型的方法集合，这里，我就不再例子中重复列出 dumpMethodSet 的代码了。

结构体类型中嵌入接口类型

在结构体类型中嵌入接口类型后，结构体类型的方法集合会发生什么变化呢？我们通过下面这个例子来看一下：

[复制代码](#)

```
1 type I interface {
2     M1()
3     M2()
4 }
5
6 type T struct {
7     I
8 }
9
10 func (T) M3() {}
11
12 func main() {
13     var t T
14     var p *T
15     dumpMethodSet(t)
16     dumpMethodSet(p)
17 }
```

运行这个示例，我们会得到以下结果：

[复制代码](#)

```
1 main.T's method set:
2 - M1
3 - M2
4 - M3
5
6 *main.T's method set:
7 - M1
8 - M2
9 - M3
```

我们可以看到，原本结构体类型 `T` 只带有一个方法 `M3`，但在嵌入接口类型 `I` 后，结构体类型 `T` 的方法集合中又并入了接口类型 `I` 的方法集合。并且，由于 `*T` 类型方法集合包括 `T` 类型的方法集合，因此无论是类型 `T` 还是类型 `*T`，它们的方法集合都包含 `M1`、`M2` 和 `M3`。于是我们可以得出一个结论：**结构体类型的方法集合，包含嵌入的接口类型的方法集合。**

不过有一种情况，你要注意一下，那就是当结构体嵌入的多个接口类型的方法集合存在交集时，你要小心编译器可能会出现错误提示。

看到这里，有同学可能会问：老师，你不是说 Go 1.14 版本解决了嵌入接口类型的方法集合有交集的情况吗？没错，但那仅限于接口类型中嵌入接口类型，这里我们说的是在结构体类型中嵌入方法集合有交集的接口类型。

这是什么意思呢？根据我们前面讲的，嵌入了其他类型的结构体类型本身是一个代理，在调用其实例所代理的方法时，Go 会首先查看结构体自身是否实现了该方法。

如果实现了，Go 就会优先使用结构体自己实现的方法。如果没有实现，那么 Go 就会查找结构体中的嵌入字段的方法集合中，是否包含了这个方法。如果多个嵌入字段的方法集合中都包含这个方法，那么我们就说方法集合存在交集。这个时候，Go 编译器就会因无法确定究竟使用哪个方法而报错，下面的这个例子就演示了这种情况：

 复制代码

```
1  type E1 interface {
2      M1()
3      M2()
4      M3()
5  }
6
7  type E2 interface {
8      M1()
9      M2()
10     M4()
11 }
12
13 type T struct {
14     E1
15     E2
16 }
17
18 func main() {
19     t := T{}
20     t.M1()
21     t.M2()
22 }
```

运行这个例子，我们会得到：

 复制代码

```
1 main.go:22:3: ambiguous selector t.M1
2 main.go:23:3: ambiguous selector t.M2
```

我们看到，Go 编译器给出了错误提示，表示在调用 `t.M1` 和 `t.M2` 时，编译器都出现了分歧。在这个例子中，结构体类型 `T` 嵌入的两个接口类型 `E1` 和 `E2` 的方法集合存在交集，都包含 `M1` 和 `M2`，而结构体类型 `T` 自身呢，又没有实现 `M1` 和 `M2`，所以编译器会因无法做出选择而报错。

那怎么解决这个问题呢？其实有两种解决方案。一是，我们可以消除 `E1` 和 `E2` 方法集合存在交集的情况。二是为 `T` 增加 `M1` 和 `M2` 方法的实现，这样的话，编译器便会直接选择 `T` 自己实现的 `M1` 和 `M2`，不会陷入两难境地。比如，下面的例子演示的就是 `T` 增加了 `M1` 和 `M2` 方法实现的情况：

 复制代码

```
1 ... ...
2 type T struct {
3     E1
4     E2
5 }
6
7 func (T) M1() { println("T's M1") }
8 func (T) M2() { println("T's M2") }
9
10 func main() {
11     t := T{}
12     t.M1() // T's M1
13     t.M2() // T's M2
14 }
```

结构体类型嵌入接口类型在日常编码中有一个妙用，就是**可以简化单元测试的编写**。由于嵌入某接口类型的结构体类型的方法集合包含了这个接口类型的方法集合，这就意味着，这个结构体类型也是它嵌入的接口类型的一个实现。即便结构体类型自身并没有实现这个接口类型的任意一个方法，也没有关系。我们来看一个直观的例子：

 复制代码

```
1 package employee
2
3 type Result struct {
4
```

```
5     Count int
6 }
7
8 func (r Result) Int() int { return r.Count }
9
10 type Rows []struct{}
11
12 type Stmt interface {
13     Close() error
14     NumInput() int
15     Exec(stmt string, args ...string) (Result, error)
16     Query(args []string) (Rows, error)
17 }
18
19 // 返回男性员工总数
20 func MaleCount(s Stmt) (int, error) {
21     result, err := s.Exec("select count(*) from employee_tab where gender=?",
22         if err != nil {
23             return 0, err
24         }
25
26     return result.Int(), nil
27 }
```

在这个例子中，我们有一个 `employee` 包，这个包中的方法 `MaleCount`，通过传入的 `Stmt` 接口的实现从数据库获取男性员工的数量。

现在我们的任务是要对 `MaleCount` 方法编写单元测试代码。对于这种依赖外部数据库操作的方法，我们的惯例是使用“伪对象（fake object）”来冒充真实的 `Stmt` 接口实现。

不过现在有一个问题，那就是 `Stmt` 接口类型的方法集合中有四个方法，而 `MaleCount` 函数只使用了 `Stmt` 接口的一个方法 `Exec`。如果我们针对每个测试用例所用的伪对象都实现这四个方法，那么这个工作量有些大。

那么这个时候，我们怎样快速建立伪对象呢？结构体类型嵌入接口类型便可以帮助我们，下面是我们的解决方案：

[复制代码](#)

```
1 package employee
2
3 import "testing"
4
5 type fakeStmtForMaleCount struct {
6     Stmt
```

```
7 }
8
9 func (fakeStmtForMaleCount) Exec(stmt string, args ...string) (Result, error)
10     return Result{Count: 5}, nil
11 }
12
13 func TestEmployeeMaleCount(t *testing.T) {
14     f := fakeStmtForMaleCount{}
15     c, _ := MaleCount(f)
16     if c != 5 {
17         t.Errorf("want: %d, actual: %d", 5, c)
18         return
19     }
20 }
```

我们为 TestEmployeeMaleCount 测试用例建立了一个 fakeStmtForMaleCount 的伪对象类型，然后在这个类型中嵌入了 Stmt 接口类型。这样 fakeStmtForMaleCount 就实现了 Stmt 接口，我们也实现了快速建立伪对象的目的。接下来我们只需要为 fakeStmtForMaleCount 实现 MaleCount 所需的 Exec 方法，就可以满足这个测试的要求了。

那说完了在结构体中嵌入接口类型的情况后，我们再来看在结构体中嵌入结构体类型会对方法集合产生什么影响。

结构体类型中嵌入结构体类型

我们前面已经学过，在结构体类型中嵌入结构体类型，为 Gopher 们提供了一种“实现继承”的手段，外部的结构体类型 T 可以“继承”嵌入的结构体类型的所有方法的实现。并且，无论是 T 类型的变量实例还是 *T 类型变量实例，都可以调用所有“继承”的方法。但这种情况下，带有嵌入类型的新类型究竟“继承”了哪些方法，我们还要通过下面这个具体的示例来看一下。

[复制代码](#)

```
1 type T1 struct{}
2
3 func (T1) T1M1() { println("T1's M1") }
4 func (*T1) PT1M2() { println("PT1's M2") }
5
6 type T2 struct{}
7
8 func (T2) T2M1() { println("T2's M1") }
9 func (*T2) PT2M2() { println("PT2's M2") }
```

```
10  type T struct {
11      T1
12      *T2
13  }
14
15  func main() {
16      t := T{
17          T1: T1{},
18          T2: &T2{},
19      }
20
21      dumpMethodSet(t)
22      dumpMethodSet(&t)
23  }
24
```

在这个例子中，结构体类型 `T` 有两个嵌入字段，分别是 `T1` 和 `*T2`，根据上一讲中我们对结构体的方法集合的讲解，我们知道 `T1` 与 `*T1`、`T2` 与 `*T2` 的方法集合是不同的：

`T1` 的方法集合包含：`T1M1`；

`*T1` 的方法集合包含：`T1M1`、`PT1M2`；

`T2` 的方法集合包含：`T2M1`；

`*T2` 的方法集合包含：`T2M1`、`PT2M2`。

它们作为嵌入字段嵌入到 `T` 中后，对 `T` 和 `*T` 的方法集合的影响也是不同的。我们运行一下这个示例，看一下输出结果：

```
1  main.T's method set:
2  - PT2M2
3  - T1M1
4  - T2M1
5
6  *main.T's method set:
7  - PT1M2
8  - PT2M2
9  - T1M1
10 - T2M1
```

[复制代码](#)

通过输出结果，我们看到了 `T` 和 `*T` 类型的方法集合果然有差别的：

类型 T 的方法集合 = T1 的方法集合 + *T2 的方法集合

类型 *T 的方法集合 = *T1 的方法集合 + *T2 的方法集合

这里，我们尤其要注意 *T 类型的方法集合，它包含的可不是 T1 类型的方法集合，而是 *T1 类型的方法集合。这和结构体指针类型的方法集合包含结构体类型方法集合，是一个道理。

讲到这里，基于类型嵌入“继承”方法实现的原理，我们基本都讲清楚了。但不知道你会不会还有一点疑惑：只有通过类型嵌入才能实现方法“继承”吗？如果我使用类型声明语法基于一个已有类型 T 定义一个新类型 NT，那么 NT 是不是可以直接继承 T 的所有方法呢？

为了解答这个疑惑，我们继续来看看 defined 类型与 alias 类型是否可以实现方法集合的“继承”。

defined 类型与 alias 类型的方法集合

Go 语言中，凡通过类型声明语法声明的类型都被称为 defined 类型，下面是一些 defined 类型的声明的例子：

 复制代码

```
1 type I interface {
2     M1()
3     M2()
4 }
5 type T int
6 type NT T // 基于已存在的类型T创建新的defined类型NT
7 type NI I // 基于已存在的接口类型I创建新defined接口类型NI
```

新定义的 defined 类型与原 defined 类型是不同的类型，那么它们的方法集合上又会有什么关系呢？新类型是否“继承”原 defined 类型的方法集合呢？

这个问题，我们也要分情况来看。

对于那些基于接口类型创建的 defined 的接口类型，它们的方法集合与原接口类型的方法集合是一致的。但对于基于非接口类型的 defined 类型创建的非接口类型，我们通过下面

例子来看一下：

 复制代码

```
1 package main
2
3 type T struct{}
4
5 func (T) M1() {}
6 func (*T) M2() {}
7
8 type T1 T
9
10 func main() {
11     var t T
12     var pt *T
13     var t1 T1
14     var pt1 *T1
15
16     dumpMethodSet(t)
17     dumpMethodSet(t1)
18
19     dumpMethodSet(pt)
20     dumpMethodSet(pt1)
21 }
```

在这个例子中，我们基于一个 defined 的非接口类型 T 创建了新 defined 类型 T1，并且分别输出 T1 和 *T1 的方法集合来确认它们是否“继承”了 T 的方法集合。

运行这个示例程序，我们得到如下结果：

 复制代码

```
1 main.T's method set:
2 - M1
3
4 main.T1's method set is empty!
5
6 *main.T's method set:
7 - M1
8 - M2
9
10 *main.T1's method set is empty!
```

从输出结果上看，新类型 T1 并没有“继承”原 defined 类型 T 的任何一个方法。从逻辑上来说，这也符合 T1 与 T 是两个不同类型的语义。

基于自定义非接口类型的 defined 类型的方法集合为空的事实，也决定了即便原类型实现了某些接口，基于其创建的 defined 类型也没有“继承”这一隐式关联。也就是说，新 defined 类型要想实现那些接口，仍然需要重新实现接口的所有方法。

那么，基于类型别名（type alias）定义的新类型有没有“继承”原类型的方法集合呢？我们还是来看一个例子：

 复制代码

```
1 type T struct{}
2
3 func (T) M1() {}
4 func (*T) M2() {}
5
6 type T1 = T
7
8 func main() {
9     var t T
10    var pt *T
11    var t1 T1
12    var pt1 *T1
13
14    dumpMethodSet(t)
15    dumpMethodSet(t1)
16
17    dumpMethodSet(pt)
18    dumpMethodSet(pt1)
19 }
```

这个例子改自之前那个例子，我只是将 T1 的定义方式由类型声明改成了类型别名，我们看一下这个例子的输出结果：

 复制代码

```
1 main.T's method set:
2 - M1
3
4 main.T's method set:
5 - M1
6
7 *main.T's method set:
```

```
8 - M1
9 - M2
10
11 *main.T's method set:
12 - M1
13 - M2
```

通过这个输出结果，我们看到，我们的 `dumpMethodSet` 函数甚至都无法识别出“类型别名”，无论类型别名还是原类型，输出的都是原类型的方法集合。

由此我们可以得到一个结论：无论原类型是接口类型还是非接口类型，类型别名都与原类型拥有完全相同的方法集合。

小结

好了，今天的课讲到这里就结束了。这一节课中，我们主要学习了类型嵌入相关的知识，类型嵌入对类型方法集合的影响，也是我们日常进行方法设计时必须要考虑到的重要因素。

类型嵌入分为两种，一种是接口类型的类型嵌入，对于接口类型的类型嵌入我们只要把握好其语义“方法集合并入”就可以了。另外一种是非接口类型的类型嵌入。通过在结构体定义中的嵌入字段，我们可以实现对嵌入类型的方法集合的“继承”。

但这种“继承”并非经典面向对象范式中的那个继承，Go 中的“继承”实际是一种组合，更具体点是组合思想下代理（delegate）模式的运用，也就是新类型代理了其嵌入类型的所有方法。当外界调用新类型的方法时，Go 编译器会首先查找新类型是否实现了这个方法，如果没有，就会将调用委派给其内部实现了这个方法的嵌入类型的实例去执行，你一定要理解这个原理。

此外，你还要牢记类型嵌入对新类型的方法集合的影响，包括：

结构体类型的方法集合包含嵌入的接口类型的方法集合；

当结构体类型 `T` 包含嵌入字段 `E` 时，`*T` 的方法集合不仅包含类型 `E` 的方法集合，还要包含类型 `*E` 的方法集合。

最后，基于非接口类型的 `defined` 类型创建的新 `defined` 类型不会继承原类型的方法集合，而通过类型别名定义的新类型则和原类型拥有相同的方法集合。

思考题

请你思考一下，下面带有类型嵌入的结构体 `S1` 与不带类型嵌入的结构体 `S2` 是否是等价的，如不等价，区别在哪里？

[复制代码](#)

```
1  type T1 int
2  type t2 struct{
3      n int
4      m int
5  }
6
7  type I interface {
8      M1()
9  }
10
11 type S1 struct {
12     T1
13     *t2
14     I
15     a int
16     b string
17 }
18
19 type S2 struct {
20     T1 T1
21     t2 *t2
22     I I
23     a int
24     b string
25 }
```

欢迎把这节课分享给更多对 Go 类型嵌入感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 方法：方法集合与如何选择receiver类型？

下一篇 加餐 | 他山之石：学习Go你还可以参考什么？

更多学习推荐

2021 最新大厂 Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

免费领取



精选留言 (10)

💬 写留言



一步

2021-12-14

S2 不是嵌入的，相当的正常书写的字段，所有没有 对应接口的方法集合



👍 5



liaomars

2021-12-13

带有类型嵌入的结构体 S1 与不带类型嵌入的结构体 S2 是否是等价的，如不等价，区别在哪里？

这两个S1与S2是不等价的，区别是：S1结构体能调用代理嵌入类型的所有方法，S2结构体是没有代理嵌入类型方法。

展开

**lesserror**

2021-12-14

Tony bai 老师，我看打印出的方法集中都有类似这样的输出：`*main.T's`。后面会讲解一下这种语法输出吗？

对于课后问题，我觉得用一下老师的 `dumpMethodSet` 函数 就能很容易看出区别了。

...

展开 ∨

共 1 条评论 >



2021-12-13

区别应该是 `t2` 有没有 `*` 吧，`S2` 的 `t2.m` 或 `t2.n` 没有取地址？

共 2 条评论 >

**扣剑书生**

2021-12-13

`S2`的方法集合肯定没有 `M1()`，没有继承

展开 ∨

**罗杰**

2021-12-13

干货满满

展开 ∨

**Geek_99b47c**

2021-12-13

`S1's method set total 1, detail method :`

-- `M1`

`S2's method set is empty`

这是为啥啊

共 1 条评论 >

**不负青春不负己**

2021-12-15

`S1` 符合预期继承了`I`接口的`M1`方法,而`S2`去没有

展开 ▾



tequ1lAneio

2021-12-14

> 结构体类型的方法集合包含嵌入的接口类型的方法集合
S1嵌入了接口I的方法集合，S1的方法集合中出现了M1，相当于实现了M1。
S2并没有嵌入接口I，所以S2的方法集合中没有接口中定义的M1。

展开 ▾



功夫熊猫

2021-12-13

type T1 = T和type T1 T两个语法有什么区别吗？

展开 ▾

共 4 条评论 >

